Programación Funcional Avanzada

QuickCheck – Verificación de Código Puro

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

QuickCheck

- Herramienta para asistir en la verificación automática de programas.
- Librería de combinadores Haskell (EDSL).
- El programador provee una *especificación* que describe **propiedades** que deben cumpliar las funciones que se desea verificar.
- `QuickCheck` *genera* casos de prueba y *comprueba* las propiedades.
  - Generación de datos de prueba controlable por el programador.
  - *Distribución* de los datos de prueba controlable por el programador.
  - Si una propiedad no se cumple, se reporta el contraejemplo.
- Aplicable a código funcional puro o código monádico transparente.

Hablaré de QuickCheck 2
No usen QuickCheck 1

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué es una *propiedad*?

Cualquier expresión lógica sobre sus argumentos

```
prop_assoc x y z = (x + y) + z == x + (y + z)
```

¿Qué es una *propiedad*?

Cualquier expresión lógica sobre sus argumentos

```
prop_assoc x y z = (x + y) + z == x + (y + z)
```

Tipos polimórficos permiten reutilizar

```
ghci> quickCheck
      (prop_assoc :: Int -> Int -> Int -> Bool)
+++ OK, passwd 100 tests.
ghci> quickCheck
      (prop_assoc :: Float -> Float -> Float -> Bool)
*** Failed! Falsifiable (after 2 tests and 1 shrink):
-2.0
0.7972123
-0.4094756
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Qué es una *propiedad*?

Cualquier expresión lógica sobre sus argumentos

```
prop_assoc x y z = (x + y) + z == x + (y + z)
```

Tipos polimórficos permiten reutilizar

```
ghci> quickCheck
      (prop_assoc :: Int -> Int -> Int -> Bool)
+++ OK, passwd 100 tests.
ghci> quickCheck
      (prop_assoc :: Float -> Float -> Float -> Bool)
*** Failed! Falsifiable (after 2 tests and 1 shrink):
-2.0
0.7972123
-0.4094756
```

Se acostumbra utilizar el prefijo `prop_`
para identificar las propiedades en el código.

Idea general

Nuestras funciones de ordenamiento

- Hace algunas semanas escribimos una función para ordenar con el algoritmo *Insertion Sort*.

```
isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []     = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                                else y : ins x ys
```

¿Cómo podemos *probar* que funciona bien?

UNIVERSIDAD SIMÓN BOLÍVAR

Idea general

Nuestras funciones de ordenamiento

- Hace algunas semanas escribimos una función para ordenar con el algoritmo *Insertion Sort*.

```
isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []     = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                                else y : ins x ys
```

  ¿Cómo podemos *probar* que funciona bien?

- Construir propiedades de la función de ordenamiento.

UNIVERSIDAD SIMÓN BOLÍVAR

Idea general

Nuestras funciones de ordenamiento

- Hace algunas semanas escribimos una función para ordenar con el algoritmo *Insertion Sort*.

```
isort :: Ord a => [a] -> [a]
isort = foldr ins []
  where ins x []     = [x]
        ins x (y:ys) = if x < y then (x:y:ys)
                                else y : ins x ys
```

  ¿Cómo podemos *probar* que funciona bien?
- Construir propiedades de la función de ordenamiento.
- Generar listas de prueba y aplicar las propiedades.

UNIVERSIDAD SIMÓN BOLÍVAR

Idea general

Propiedades de la función de ordenamiento

- Idempotente – ordenar lo ordenado es lo mismo que ordenar una vez

```
prop_idempotent xs = isort (isort xs) == isort xs
```

UNIVERSIDAD SIMÓN BOLÍVAR

Idea general

Propiedades de la función de ordenamiento

- Idempotente – ordenar lo ordenado es lo mismo que ordenar una vez

```
prop_idempotent xs = isort (isort xs) == isort xs
```

- Después de ordenar, la lista debe estar ordenada

```
prop_ordered xs = ordered $ isort xs
  where ordered []       = True
        ordered [x]      = True
        ordered (x:y:ys) = x <= y && ordered (y:ys)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Idea general

Propiedades de la función de ordenamiento

- Después de ordenar, el primer elemento ha de ser el mínimo

```
prop_minimum xs =
  not (null xs) ==> head (isort xs) == minimum xs
```

UNIVERSIDAD SIMÓN BOLÍVAR

Idea general

Propiedades de la función de ordenamiento

- Después de ordenar, el primer elemento ha de ser el mínimo

```
prop_minimum xs =
  not (null xs) ==> head (isort xs) == minimum xs
```

- Si se ordena una concatenación, el mínimo del resultado es el mínimo entre los mínimos de ambas listas.

```
prop_append xs ys =
  not (null xs) ==>
    not (null ys) ==>
      head (isort (xs++ys)) == min (minimum xs)
                                   (minimum ys)
```

UNIVERSIDAD SIMÓN BOLÍVAR

QuickCheck en acción

- Las propiedades son funciones que acompañan al módulo.
- Se aprovecha la librería `QuickCheck`.

```
import Test.QuickCheck
```

- Se carga el módulo en `ghci` y se prueba manualmente

```
ghci> quickCheck prop_idempotent
+++ OK, passed 100 tests.
ghci> quickCheck prop_append
+++ OK, passed 100 tests.
ghci> quickCheck prop_minimum
+++ OK, passed 100 tests.
ghci> quickCheck prop_ordered
+++ OK, passed 100 tests.
```

UNIVERSIDAD SIMÓN BOLÍVAR

Controlando QuickCheck

Párametros de ejecución

- El tipo Args se combina con la función quickCheckWith.

```
ghci> stdArgs
Args { replay = Nothing, maxSuccess = 100,
       maxDiscard = 500, maxSize = 100 }
ghci> quickCheckWith
      (stdArgs { maxSuccess = 1000 }) prop_ordered
+++ OK, passed 1000 tests.
```

- Args controla cuántos casos generar y cuan complejos
  - replay – generador de números al azar para pruebas repetibles.
  - maxSuccess – cuántos éxitos para concluir.
  - maxDiscard – cuántos descartes para concluir.
  - maxSize – cuán *grande* cada caso.

UNIVERSIDAD SIMÓN BOLÍVAR

Pruebas en *batch*

- Pruebas manuales.
  - Utilidad limitada a depuración.
  - Requieren un humano – preferiblemente pensante.
  - Cansan, aburren y por eso fracasan.

Pruebas en *batch*

- Pruebas manuales.
  - Utilidad limitada a depuración.
  - Requieren un humano – preferiblemente pensante.
  - Cansan, aburren y por eso fracasan.
- Pruebas automáticas.
  - Cubrir *todo* el módulo.
  - Todo el día, todos los días, cuantas veces quieras.
  - Asociarlas al *commit* en el VCS.

UNIVERSIDAD SIMÓN BOLÍVAR

Pruebas en *batch*

- Pruebas manuales.
  - Utilidad limitada a depuración.
  - Requieren un humano – preferiblemente pensante.
  - Cansan, aburren y por eso fracasan.
- Pruebas automáticas.
  - Cubrir *todo* el módulo.
  - Todo el día, todos los días, cuantas veces quieras.
  - Asociarlas al *commit* en el VCS.
- ¿Cómo automatizarlas?
  - El programa `quickCheck` – buscarlo en Hackage.

```
$ runhaskell quickCheck.hs testing.hs
```

  - Escribir su propio programa de pruebas.

UNIVERSIDAD SIMÓN BOLÍVAR

Controlando `QuickCheck`

Generando datos de prueba

- Propiedades cuantificados universalmente de forma implícita – ¿cómo generar ese "para todo" de forma práctica?
- Definición de generadores

```
class Arbitrary a where
  arbitrary :: Gen a
```

  - Se proveen instancias predefinidas – `Bool`, `Int`, tuplas, listas, ...
  - `Gen` es un `Monad` `State` que maneja la generación de números al azar.
  - Basta proveer una instancia para generar datos arbitrarios.

UNIVERSIDAD SIMÓN BOLÍVAR

Funciones de apoyo

- Usar `arbitrary` en el contexto `Gen a` – resultados en `IO`.

```
sample' :: Gen a -> IO [a]
```

- Tomar "muestras" de un generador particular.

```
sample' :: Gen a -> IO [a]

ghci> sample' (arbitrary :: Gen Int)
[-1,1,2,-4,4,-9,12,-10,53,115,212]

ghci> take 3 `liftM`
              sample' (arbitrary :: Gen (Int,Bool))
[(1,False),(-1,True),(-2,False)]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Funciones de apoyo

- Usar `arbitrary` en el contexto `Gen a` – resultados en `IO`.

```
sample' :: Gen a -> IO [a]
```

- Tomar "muestras" de un generador particular.

```
sample' :: Gen a -> IO [a]

ghci> sample' (arbitrary :: Gen Int)
[-1,1,2,-4,4,-9,12,-10,53,115,212]

ghci> take 3 `liftM`
              sample' (arbitrary :: Gen (Int,Bool))
[(1,False),(-1,True),(-2,False)]
```

Aprovechar que `Gen` es un Monad

UNIVERSIDAD SIMÓN BOLÍVAR

Funciones de apoyo

De valores puros al Monad `Gen`

- `choose` – generar valores en un rango específico.

```
choose :: System.Random.Random a => (a,a) -> Gen a

ghci> sample' $ choose (0,42)
[30,31,19,41,0,7,33,41,5,7,6]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Funciones de apoyo

De valores puros al Monad `Gen`

- `choose` – generar valores en un rango específico.

```
choose :: System.Random.Random a => (a,a) -> Gen a

ghci> sample' $ choose (0,42)
[30,31,19,41,0,7,33,41,5,7,6]
```

- `elements` – algún valor de una lista específica.

```
elements :: [a] -> Gen a

ghci> sample' $ elements "abcde"
"cdbabcdddee"
```

UNIVERSIDAD SIMÓN BOLÍVAR

Combinadores de Generación

Una vez que se tiene `Gen a`

- `listOf` – lista posiblemente vacía.

```
listOf :: Gen a -> Gen [a]

ghci> sample' $ listOf $ (arbitrary :: Gen Int)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Combinadores de Generación

Una vez que se tiene `Gen a`

- `listOf` – lista posiblemente vacía.

```
listOf :: Gen a -> Gen [a]

ghci> sample' $ listOf $ (arbitrary :: Gen Int)
```

- `listOf1` – lista no vacía.

```
ghci> sample' $ listOf1 $ (arbitrary :: Gen Int)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Combinadores de Generación

Una vez que se tiene `Gen a`

- `listOf` – lista posiblemente vacía.

```
listOf :: Gen a -> Gen [a]

ghci> sample' $ listOf $ (arbitrary :: Gen Int)
```

- `listOf1` – lista no vacía.

```
ghci> sample' $ listOf1 $ (arbitrary :: Gen Int)
```

- `vectorOf n` – lista de *n* elementos.

```
vectorOf :: Int -> Gen a -> Gen [a]

ghci> sample' $ vectorOf 3 $ (arbitrary :: Gen Int)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Aún más control

Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a

suchThat (arbitrary :: Gen Int) even
```

Aún más control

Generadores a partir de generadores

- suchThat – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a

suchThat (arbitrary :: Gen Int) even
```

- oneof – selecciona uno de los generadores con probabilidad uniforme.

```
oneof :: [Gen a] -> Gen a

oneof [return "foo", return "bar", return "baz"]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Aún más control

Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a

suchThat (arbitrary :: Gen Int) even
```

- `oneof` – selecciona uno de los generadores con probabilidad uniforme.

```
oneof :: [Gen a] -> Gen a

oneof [return "foo", return "bar", return "baz"]
```

- `frequency` – selecciona con probabilidad relativa según el peso, i.e.

```
frequency :: [(Int,Gen a)] -> Gen a

frequency [(2,return True),(1,return False)]
```

genera `True` 66 % de las veces...

UNIVERSIDAD SIMÓN BOLÍVAR

Aún más control

Generadores a partir de generadores

- `suchThat` – tal que cumpla el predicado.

```
suchThat :: Gen a -> (a -> Bool) -> Gen a

suchThat (arbitrary :: Gen Int) even
```

- `oneof` – selecciona uno de los generadores con probabilidad uniforme.

```
oneof :: [Gen a] -> Gen a

oneof [return "foo", return "bar", return "baz"]
```

- `frequency` – selecciona con probabilidad relativa según el peso, i.e.

```
frequency :: [(Int,Gen a)] -> Gen a

frequency [(2,return True),(1,return False)]
```

genera `True` 66 % de las veces...

Sólo es necesario construir el generador...

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo construir un Generador?

Hay que definirlos manualmente

- Basta instanciar la clase `Arbitrary`

```
instance Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

  - El programador provee `arbitrary` usando los combinadores.
  - `shrink` – ¿cómo "simplificar" un contraejemplo?
  - La implantación automática de `shrink` es la lista vacía – hace que contraejemplos sean *fatales*, impidiendo reintentar.

- Es una clase que no puede ser derivada automáticamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo generadores

Los casos sencillos

- Para tipos enumerados, es realmente trivial

```
data MetaSyn = Foo | Bar | Baz | Qux
             deriving (Show,Eq)

instance Arbitrary MetaSyn where
  arbitrary = elements [Foo,Bar,Baz,Qux]
```

Construyendo generadores

Los casos sencillos

- Para tipos enumerados, es realmente trivial

```
data MetaSyn = Foo | Bar | Baz | Qux
             deriving (Show,Eq)

instance Arbitrary MetaSyn where
  arbitrary = elements [Foo,Bar,Baz,Qux]
```

- Que nos permite hacer

```
ghci> sample' $ (arbitrary :: Gen MetaSyn)
[Bar,Foo,Bar,Qux,Foo,Foo,Baz,Foo,Baz,Baz,Foo]
```

Basta cambiar `elements` por `frequency`
si se necesita alterar las probabilidades.

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo generadores

. . . con mucho cuidado!

- Un tipo polimórfico para árboles binarios

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
            deriving (Show,Eq)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo generadores

. . . con mucho cuidado!

- Un tipo polimórfico para árboles binarios

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
            deriving (Show,Eq)
```

- Primer intento – expresión directa

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = oneof [ liftM  Leaf   arbitrary,
                        liftM3 Branch tree
                                      arbitrary
                                      tree ]
```

Construyendo generadores

. . . con mucho cuidado!

- Un tipo polimórfico para árboles binarios

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
            deriving (Show,Eq)
```

- Primer intento – expresión directa

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = oneof [ liftM  Leaf    arbitrary,
                        liftM3 Branch  tree
                                       arbitrary
                                       tree ]
```

- Probamos

```
ghci> sample' $ (arbitrary :: Gen (Tree Int))
```

Meh. . . demasiados árboles triviales

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo un generador

. . . con mucho cuidado!

- Segundo intento – introducimos un sesgo para árboles "complicados"

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = frequency [
                  (1,liftM  Leaf   arbitrary),
                  (3,liftM3 Branch tree
                                   arbitrary
                                   tree) ]
```

Construyendo un generador

. . . con mucho cuidado!

- Segundo intento – introducimos un sesgo para árboles "complicados"

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = frequency [
                  (1,liftM  Leaf   arbitrary),
                  (3,liftM3 Branch tree
                                   arbitrary
                                   tree) ]
```

- Probamos

```
ghci> sample' $ (arbitrary :: Gen (Tree Int))
```

UNIVERSIDAD SIMÓN BOLÍVAR

Construyendo un generador

. . . con mucho cuidado!

- Segundo intento – introducimos un sesgo para árboles "complicados"

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = frequency [
                  (1,liftM  Leaf   arbitrary),
                  (3,liftM3 Branch tree
                                   arbitrary
                                   tree) ]
```

- Probamos

```
ghci> sample' $ (arbitrary :: Gen (Tree Int))
```

FUUUUUUUUUUUU ¡Demasiado grandes!
¿Serán infinitos?

Construyendo un generador

. . . con mucho cuidado!

- Tercer intento – sesgo y control de tamaño

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = sized tree'
         tree' 0           = liftM Leaf arbitrary
         tree' n | n > 0 = frequency [
                   (1,liftM  Leaf   arbitrary),
                   (3,liftM3 Branch subtree
                                    arbitrary
                                    subtree) ]
             where subtree = tree' (n `div` 2)
```

Construyendo un generador

. . . con mucho cuidado!

- Tercer intento – sesgo y control de tamaño

```
instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = tree
   where tree = sized tree'
         tree' 0           = liftM Leaf arbitrary
         tree' n | n > 0 = frequency [
                   (1,liftM  Leaf   arbitrary),
                   (3,liftM3 Branch subtree
                                    arbitrary
                                    subtree) ]
             where subtree = tree' (n `div` 2)
```

- `sized :: (Int -> Gen a) -> Gen a` – tamaño según `stdArgs`.
- Garantizamos terminación – tamaño eventualmente es cero.
- Mejoramos distribución – sesgo hacia árboles "complicados".

UNIVERSIDAD SIMÓN BOLÍVAR

Un ejemplo más variado

```
data RegExp = Lambda
            | Symbol Char
            | Concat RegExp RegExp
            | Union RegExp RegExp
            | Star RegExp
            deriving (Show,Eq)
```

- ¡Expresiones regulares!
- Enfatizar el uso de casos base – `Lambda` y `Symbol`
- Controlar `Star (Star (Star ...))`

UNIVERSIDAD SIMÓN BOLÍVAR

Un ejemplo más variado

```
re :: Int -> Int -> Gen RegExp
re stars 0 = frequency [
    (1, return Lambda),
    (3, Symbol `liftM` choose ('a','z'))
]

re stars n = frequency [
    (3, return Lambda),
    (8, Symbol `liftM` choose ('a','z')),
    (4, liftM2 Concat (re stars (n `div` 2))
                      (re stars (n `div` 2))),
    (4, liftM2 Union  (re stars (n `div` 2))
                      (re stars (n `div` 2))),
    (stars, Star `liftM` (re (n-1) (n `div` 2)))
]

instance Arbitrary RegExp where
  arbitrary = sized (re 1)
```

Clasificando las pruebas

Por nombre

¿Habrá muchos casos “triviales”?

Clasificando las pruebas

Por nombre

¿Habrá muchos casos “triviales”?

```
prop_idempotent' xs =
  classify (null xs)        "trivial" $
  classify (not (null xs)) "mejor"   $
  isort (isort xs) == isort xs
```

UNIVERSIDAD SIMÓN BOLÍVAR

Clasificando las pruebas

Por nombre

¿Habrá muchos casos “triviales”?

```
prop_idempotent' xs =
  classify (null xs)        "trivial" $
  classify (not (null xs)) "mejor"   $
  isort (isort xs) == isort xs
```

. . . al ejecutarlo

```
ghci> quickCheck prop_idempotent'
+++ OK, passed 100 tests:
95% mejor
 5% trivial
```

Justificación para ajustar distribuciones.

UNIVERSIDAD SIMÓN BOLÍVAR

Clasificando las pruebas

Por distribución

¿Qué longitudes tienen las listas de prueba?

Clasificando las pruebas

Por distribución

¿Qué longitudes tienen las listas de prueba?

```
prop_idempotent'' xs = collect (length xs `div` 10) $
  isort (isort xs) == isort xs
```

Clasificando las pruebas

Por distribución

¿Qué longitudes tienen las listas de prueba?

```
prop_idempotent'' xs = collect (length xs `div` 10) $
  isort (isort xs) == isort xs
```

...al ejecutarlo

```
ghci> quickCheck prop_idempotent''
+++ OK, passed 100 tests:
39% 0
16% 1
14% 2
12% 3
 7% 4
 5% 6
 3% 9
 3% 8
 1% 7
```

¿Y si quiero ver *todas* las pruebas?

```
prop_sick x = collect x $ ...
```

¿Y si quiero ver *todas* las pruebas?

```
prop_sick x = collect x $ ...
```

Busca ayuda profesional. . .

UNIVERSIDAD SIMÓN BOLÍVAR

Cobertura de Ejecución

¿Qué es lo que estamos probando exactamente?

- El código de un programa puede ser
  - **Inalcanzable** – cuando no es utilizado, bien sea por que es una parte no utilizada de una librería o porque el flujo de ejecución jamás podrá pasar por allí.
  - **Alcanzable** – cuando, en principio, el programa puede pasar por allí.
  - Los compiladores son capaces de eliminar el primero.
  - Probar lo que quede es responsabilidad del programador.

UNIVERSIDAD SIMÓN BOLÍVAR

Cobertura de Ejecución

¿Qué es lo que estamos probando exactamente?

- El código de un programa puede ser
  - **Inalcanzable** – cuando no es utilizado, bien sea por que es una parte no utilizada de una librería o porque el flujo de ejecución jamás podrá pasar por allí.
  - **Alcanzable** – cuando, en principio, el programa puede pasar por allí.
  - Los compiladores son capaces de eliminar el primero.
  - Probar lo que quede es responsabilidad del programador.
- En el curso de una corrida, el código alcanzable puede ser
  - **Cubierto** – cuando en efecto se ejecuta.
  - **No cubierto** – cuando no se ejecuta.

Código **no cubierto** es código no probado.
Danger, Will Robinson!

UNIVERSIDAD SIMÓN BOLÍVAR

Análisis de cobertura

- **HPC** (*Haskell Program Coverage*) – herramienta integrada con el compilador `ghc` capaz de identificar código no cubierto.
- Usarlo durante las fases de desarrollo y prueba.
  - Compilar el programa con el flag `-fhpc`.
  - Ejecutar el programa – mientras más veces, mejor.
  - `hpc report` – reporte breve.
  - `hpc markup` – reporte HTML con anotaciones sobre el *código fuente*.
- El reporte de HPC
  - Presenta código no cubierto hasta el nivel de **subexpresiones**.
  - Resalta condicionales que han sido siempre ciertos o siempre falsos – se denominan *brazos inertes* y son oportunidad de mejora.

UNIVERSIDAD SIMÓN BOLÍVAR

Así se hace en la práctica...

Escribimos nuestro programa de prueba

```
main = do
  quickCheck
    (prop_idempotent :: [Int] -> Bool)
  quickCheck
    (prop_minimum :: [Int] -> Property)
  quickCheck
    (prop_ordered :: [Int] -> Bool)
  quickCheck
    (prop_append :: [Int] -> [Int] -> Property)
```

Las firmas son necesarias si no se pueden inferir del contexto.

UNIVERSIDAD SIMÓN BOLÍVAR

En la práctica

Compilar para usar HPC

- Compilamos con las opciones necesarias

```
$ ghc -fhpc -fforce-recomp --make testing.hs
```

- Corremos el programa tantas veces como se desee.
- Archivo `.tix` *acumula* las medidas de cobertura – eliminarlo para comenzar desde cero.

¡No olvide recompilar **sin** `-fhpc` antes de pasar a producción!

UNIVERSIDAD SIMÓN BOLÍVAR

Evaluando los resultados

Desde la línea de comandos

```
$ hpc report testing
 53% expressions used (77/144)
 50% boolean coverage (1/2)
      0% guards (0/1), 1 unevaluated
    100% 'if' conditions (1/1)
    100% qualifiers (0/0)
 77% alternatives used (7/9)
 40% local declarations used (2/5)
 42% top-level declarations used (6/14)
```

Evaluando los resultados

A través de un navegador

Generar los documentos HTML resumen.

```
$ hpc markup testing
Writing: Main.hs.html
Writing: hpc_index.html
Writing: hpc_index_fun.html
Writing: hpc_index_alt.html
Writing: hpc_index_exp.html
```

El índice `hpc_index.html` muestra un resumen.

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	42%	6/14	77%	7/9	53%	77/144
Program Coverage Total	42%	6/14	77%	7/9	53%	77/144

UNIVERSIDAD SIMÓN BOLÍVAR

Mejores prácticas

- Escriba pruebas para todas las funciones del módulo.
- Escriba un programa que ejercite todas las pruebas del módulo de manera automática.
- Ejecute el programa de prueba bajo HPC.
- Intente alcanzar 100 % de cobertura – puede ser muy difícil.
- Los contraejemplos encontrados por QuickCheck ayudan a identificar defectos en la implantación
  - En ocasiones son difíciles de comprender.
  - "Encogerlos" sintácticamente – `shrink :: a -> [a]`.
  - Ejemplos simples primero, complejos después.

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Documentación de QuickCheck
- Introduction to QuickCheck2 en Haskell WiKi
- Haskell Program Coverage

QuickCheck 2
(se lee “dos”)

UNIVERSIDAD SIMÓN BOLÍVAR